

DISTEA: Efficient Dynamic Impact Analysis for Distributed Systems

Haipeng Cai and Douglas Thain
University of Notre Dame, Indiana, USA
email: {hcai|dthain}@nd.edu

Abstract—Dynamic impact analysis is a fundamental technique for understanding the impact of specific program entities, or changes to them, on the rest of the program for concrete executions. However, existing techniques are either inapplicable or of very limited utility for distributed programs running in multiple concurrent processes. This paper presents DISTEA, a technique and tool for dynamic impact analysis of distributed systems. By partially ordering distributed method-execution events and inferring causality from the ordered events, DISTEA can predict impacts propagated both within and across process boundaries. We implemented DISTEA for Java and applied it to four distributed programs of various types and sizes, including two enterprise systems. We also evaluated the precision and practical usefulness of DISTEA, and demonstrated its application in program comprehension, through two case studies. The results show that DISTEA is highly scalable, more effective than existing alternatives, and instrumental to understanding distributed systems and their executions.

I. INTRODUCTION

Program changes drive the evolution of software systems, yet also pose threats to their quality and reliability [1]. Thus, it is crucial to understand potential consequences of those changes even *before* applying them to candidate program locations. To accomplish this task, developers need to perform impact analysis [2]–[4] with respect to those locations, an integral step of modern software development process [5]. In particular, for developers working with specific operational profiles of the program, dynamic impact analysis [2], [6] is an attractive option as it narrows down the search space of such impacts to the concrete context of those profiles.

During the past two decades, research on dynamic impact analysis has been extensively invested [6], resulting in a rich and diverse set of relevant techniques and tools (e.g., [7]–[10]). However, most of existing such approaches mainly address sequential programs only, with much less targeting concurrent yet centralized software [11]–[15] while very few applicable to distributed systems. On the other hand, to accommodate the increasingly demanding performance and scalability needs of today’s computation tasks, more distributed systems than centralized ones are being deployed, raising an urgent call for technical supports, including impact analysis, for effective maintenance and evolution of those systems [11], [16]–[18].

Code analysis techniques for distributed systems were explored since early on [19]–[21] and ratcheted up recently [22]–[25], largely focusing on detailed analysis of program dependencies. However, the majority of these approaches were designed only for procedural programs [23]. For distributed object-oriented programs, *backward* dynamic slicing algorithms have been developed, yet it is still unclear whether they can work with real-world systems [22], [23]. And for impact analysis, *forward* slicing would be needed. Nevertheless, the fine-grained (statement-level) analysis used

by slicing would be overly heavyweight for impact analysis commonly adopted at method level [2], [6]–[9].

Unfortunately, developing an efficient dynamic impact analysis for distributed systems remains challenging. One major difficulty lies in the lack of explicit invocations or references among decoupled components in such systems [12], [16], [26], whereas traditional approaches usually rely on those explicit information to compute dependencies for impact prediction. Lately, various dependence-analysis techniques other than slicing have also been proposed [17], [26], [27]. While efficient for *static* impact analysis, these approaches are limited to systems of special type such as distributed *event-based* systems (DEBS) [28], or rely on specialized language extensions like EventJava [29]. Other approaches are potentially applicable in a wider scope, yet they depend on information not always available, such as execution logs of particular pattern [30], or suffer from overly-coarse granularity (e.g., class-level) [17], [27], [30] and/or unsoundness [31], in addition to imprecision, of their analysis results.

In this paper, we present DISTEA, a dynamic impact analysis approach for commonly deployed distributed systems where components communicate via socket-based message passing.¹By exploiting the happens-before relations [32] among distributed method-execution events, DISTEA predicts impacts of one method on others of a given system both within and across its concurrent processes. Akin to the execute-after-sequences (EAS) technique [8], our approach offers results that are safe relative to the concrete executions utilized with high efficiency, while relying on neither well-defined inter-component interfaces nor message-type specifications as needed by peer approaches (for DEBS).

We evaluate DISTEA on four distributed Java programs, including two enterprise systems, and demonstrate that it is able to work with large distributed systems with both blocking and non-blocking (e.g., selector-based [33]) communications. In the absence of peer techniques directly comparable to ours, we take a coverage-based approach [34], which reports as impacted all methods covered in the utilized executions, as a safe baseline alternative, and measure the effectiveness of DISTEA against it. The results show that DISTEA can greatly reduce the size of potential impacts to be inspected, by 36% on average, relative to the baseline, at the mean cost of 50 seconds to finish the one-time static analysis and three seconds to answer a query, with a runtime overhead of 11%.

Since there is no automatic approach available to us for computing ground-truth impacts either, we manually evaluate the precision of DISTEA on five randomly selected cases in a case study. Also, we explore the usefulness of DISTEA in program comprehension in a second case study. Our results suggest that developers using DISTEA may expect an

¹We distinguish components as such that each runs in a separate process.

```

1  public class C {
2      Socket csock = null;
3      public C(String host, int port) {
4          csock = new Socket(host, port); }
5      void shuffle(String s) {...}
6      char compute(String s) {
7          shuffle(s);
8          csock.writeChars(s);
9          return sock.readChar(); }
10     public static int main(String[] a) {
11         C c = new C("localhost", 2345);
12         System.out.println( c.compute(a[0]) );
13         return 0; } }
14     public class S {
15         Socket ssock = null;
16         public S(int port) {
17             ssock = new Socket(port);
18             ssock.accept(); }
19         char getMax(String s) {...}
20         void serve() {
21             String s = ssock.readLine();
22             char r = getMax(s);
23             ssock.writeChar(r); }
24         public static int main(String[] a) {
25             S s = new S(2345);
26             return s.serve(); } }

```

Fig. 1: An example distributed program E consisting of two components: C (client) and S (server).

average precision of about 60% and considerable benefits for understanding distributed programs and executions.

The main contributions of this work include:

- A dynamic impact analysis, DISTEA, for distributed systems where concurrent processes communicate via socket-based message passing (Section III).
- An implementation of DISTEA for Java working with large enterprise distributed systems with both blocking and non-blocking communications (Section IV).
- An empirical study of DISTEA showing its promising effectiveness and scalability (Section V).
- Two in-depth case studies of DISTEA showing its practical usefulness for impact analysis and benefits for distributed-program understanding (Section VI).

II. MOTIVATION AND BACKGROUND

In this section, we first present a usage scenario of dynamic impact analysis that motivates our development of DISTEA. Then, we give necessary background on techniques underlying the design of DISTEA.

A. Motivating Example

When maintaining and evolving a distributed program which consists of multiple components, the developer needs to understand potential change effects not only in the component where the change is proposed, but also those in all other components. To achieve better flexibility and scalability, these components are usually loosely coupled or entirely decoupled as a result of implicit invocations among them realized via socket-based message passing, which, however, reduces the utility of existing impact analysis to a very limited extent.

Consider the example program E shown in Figure 1, which consists of two components: a server and a client, implemented in classes S and C , respectively. The client simply retrieves the largest character in a given string by sending the task to the

server, which finishes the task and sends the result back to the client. Suppose now the developer proposes to apply a new algorithm in the $S::getMax$ method as part of an upgrade plan for the server and, thus, needs to determine which other parts of the program may have to be changed as well. Having an available set I of inputs, the developer wants to perform a dynamic impact analysis to get a quick but safe estimation on potential impacts of the candidate change with respect to I .

At first glance, it seems that the developer has many options (e.g., [8]–[10]) to accomplish this task. Unfortunately, it soon turns out that those existing options have merely quite limited utility in this context. Since there is no explicit dependencies between S and C , existing dynamic impact analysis would predict impacts within the *local* component (i.e., where the changes are located; S in this case) only. In consequence, the developer would have to ignore impacts in *remote* components (C in this case), or make a worst-case assumption that all methods in remote components are to be impacted.

As illustrated by this example, the distributed system we address in this work is one in which components located at networked computers communicate and coordinate their actions only by passing messages [35]: The components run concurrently in multiple processes *without a global clock*.

B. Dynamic Impact Analysis

Typically, a dynamic impact analysis technique inputs a program P , an input set I , and a *query set* M (the set of methods for which impacts are to be queried), and outputs an *impact set* (the set of methods in P potentially impacted) of M when running I . One representative such technique is based on the execute-after-sequences (EAS) [8], which computes impacts from the execution order of methods. Given a query c , EAS considers all methods that execute after c as potentially affected by c or by any changes to it.

To find the method execution order, EAS records two main method-execution events using two integers for each method m : the first time m is entered and the last time program control is returned into m . Then, the analysis infers the execute-after relations according to the occurrence time of those events. In presence of multi-threaded executions, EAS monitors also method returns and treats them as returned-into events. For the concrete set of executions, no methods that never executed after the query c can be impacted by it; thus, the results produced by EAS are safe (i.e., of 100% recall) relative to those executions. However, an execute-after relation does not always lead to an impact relation since a method may execute after the query yet has no any dependence on that query; thus, EAS is imprecise due to its conservative nature [36].

On the other hand, the need for maintaining only little information (i.e., the two integers per method) enables the high efficiency of EAS. Therefore, despite of its known imprecision, impact analysis using execute-after relations like EAS remains a viable option, especially for users who desire getting a safe approximation of impacts quickly, such as the developer in the above example scenario. In fact, to the best of our knowledge, EAS is still the most efficient dynamic impact analysis to this date [9], [10], [37]. Thus, as the first attempt exploring efficient dynamic impact analysis for distributed systems, we start with an EAS-based approach in this work.

Nonetheless, EAS itself does not work with distributed programs. One may attempt to first apply this technique to each component independently, generate the execute-after

sequences for each (process), and then compute impacts for the entire system by referring to all the event sequences. Unfortunately, different components often run asynchronously and on machines of different physical clocks [18], [30], easily leading to incorrect order of method executions with respect to all processes, hence erroneous impact sets.

C. Timing in Distributed Systems

Different approaches exist to manage the timing in distributed systems [32], [38], [39], of which the one by Lamport [32] maintains a logical clock per process to partially order distributed events over all processes with a simple algorithm, which we refer to as the *Lamport TimeStamping* (LTS) algorithm. The LTS approach first defines a logical clock C_i for each process P_i , which is a function that assigns a number $C_i(a)$ to an event a in P_i . Based on this definition, an event a happened before another event b if the number assigned to a is less than that assigned to b , or formally

$$a \rightarrow b \implies C(a) < C(b) \quad (1)$$

which is called the *clock condition*. Then, to maintain the clock condition during system executions, the following rules [32] should be observed by each process:

- Each process P_i increments C_i between any two successive events that happened in P_i .
- If event a is that process P_i sends a message m , then the message contains a timestamp $T_m = C_i(a)$.
- When a process P_j receives a message m , it sets C_j greater than or equal to its current value and greater than T_m .

For our EAS-based approach to impact analysis for distributed systems, the LTS algorithm can be utilized to preserve the partial ordering of distributed events across multiple processes running on separated machines. Furthermore, this ordering would enable inferring causality between methods hence the computation of impacts of one method on others both within and across system components.

III. APPROACH

To achieve an efficient dynamic impact analysis for distributed programs, DISTEA utilizes only lightweight runtime information on method execution order. We first present the fundamentals underlying our approach, including the definition of method events used by DISTEA and its rationale for impact prediction. Then, we give an overview and illustration on the inner workings of DISTEA followed by details on the analysis algorithms.

A. Fundamentals

1) *Method-Execution Events*: In the general context of distributed systems, an event is defined as any happening of interest observable from within a computer [32]. More specifically, events in a DEBS are often expressed as messages transferred among system components and defined by a set of attributes [17], [28]. In contrast, while it also deals with message passing in distributed systems, DISTEA neither makes any assumption nor reasons about the structure or content of the messages. Particularly, for dynamic impact analysis, DISTEA monitors and utilizes two major classes of events as defined below:

- **Communication Event.** A communication event E_C is an occurrence of message transfer between two components $c1$ and $c2$, denoted as $E_C(c1, c2)$ if $c1$ initiates E_C which attempts to reach $c2$. Further, according to the direction

of message flow, we distinguish two major subcategories of such events: *sending a message* to a component and *receiving a message* from a component.

- **Internal Event.** An internal event E_I is an occurrence of method execution within a component c , denoted as $E_I(c)$. Further, we differentiate three subcategories of internal events: *entering a method*, *returning from a method*, and *returning into a method*, denoted as m_e , m_x , and m_i , respectively, for the relevant method m .

For internal events, we capture both the return and returned-into events for each method. However, we distinguish them during static analysis only and treat them equally in the monitoring algorithm (Section III-C2). The reason is to correctly identify execute-after relations from interleaving method executions in multiple threads as detailed in [8].

2) *Impact Inference*: One challenge to developing DISTEA is to infer the execute-after relations in the presence of asynchronous events over multiprocess concurrent executions. Fortunately, maintaining a logical notion of time per process to discover just a partial ordering of method-execution events suffices for that inference required in DISTEA.

In essence, the execute-after (EA) relation between any two methods can be semantically deduced from the happens-before relation between relevant internal events of corresponding methods; and the partial ordering of the internal events reveals such happens-before relations [32]. Formally, given two executed methods $m1$ and $m2$, we have

$$m1_e \prec m2_x \vee m1_e \prec m2_i \implies EA(m2, m1) \quad (2)$$

where \prec is the *happens-before* relation. Without loss of generality, $m1_e \prec m2_x$ and $m1_e \prec m2_i$ both imply that “ $m2$ executes after $m1$, thus $m2$ may be affected by $m1$ or by any changes to $m1$ ”, hence the execute-after relation $EA(m2, m1)$ between $m1$ and $m2$.

Based on the above inference, for a given query c , computing the impact set $IS(c)$ of c is reduced to retrieving methods, from multiprocess method-execution event sequences, that satisfy the partial ordering of internal events of candidate methods as follows:

$$IS(c) = \{m \mid c_e \prec m_i \vee c_e \prec m_x\} \quad (3)$$

Note that only internal events are directly used for impact inference, while communication events are utilized to maintain the partial ordering of internal events in all processes.

B. DISTEA Overview

1) *Process*: The overall process flow of our technique is depicted in Figure 2, where the three primary inputs are the system D under analysis, a set I of program inputs for D , and a query set M . An optional input, a message-passing API list L can also be specified to help DISTEA identify program locations where probes for communication events should be instrumented (as detailed in Section IV). The output of DISTEA is a set of potential impacts of M computed from the given inputs in four steps as annotated in the figure.

The *first step* performs the static analysis in DISTEA, where the input program D is instrumented for both monitoring method-execution events and synchronizing logical clocks among concurrent processes, and the instrumented version D' of D is produced. Then, the *second step* executes D' on the given input set I , during which internal events are produced and time-stamped by means of communication

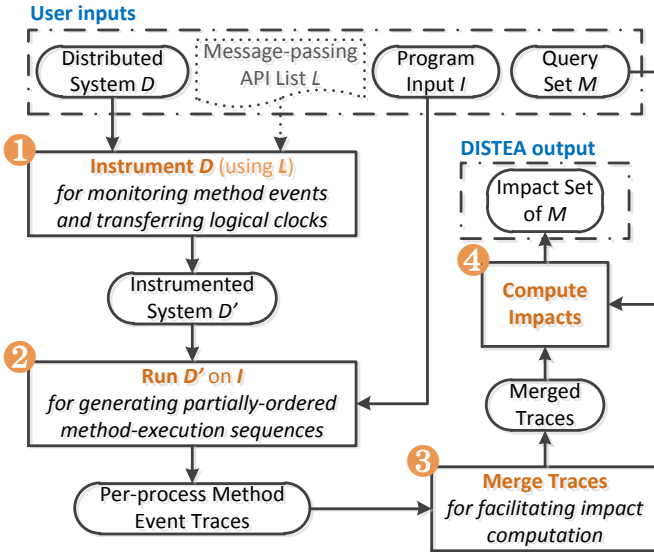


Fig. 2: The overall process flow of DISTEA, where the numbered steps are detailed in Section III-B1.

TABLE I: A FULL METHOD-EXECUTION EVENT SEQUENCE OF THE EXAMPLE PROGRAM *E*.

Server process		Client process	
Method Event	Timestamp	Method Event	Timestamp
$S::main_e$	0	$C::main_e$	0
$S::init_e$	1	$C::init_e$	1
$S::init_i$	2	$C::init_i$	2
$S::init_x$	3	$C::init_x$	3
$S::main_i$	4	$C::main_i$	4
$S::serve_e$	5	$C::compute_e$	5
$E_c(C,S)$	-	$C::shuffle_e$	6
$S::getMax_e$	10	$C::shuffle_i$	7
$S::getMax_i$	11	$C::shuffle_x$	8
$S::getMax_x$	12	$C::compute_i$	9
$S::serve_i$	13	$E_c(S,C)$	-
$E_c(S,C)$	-	$E_c(S,C)$	-
$S::serve_x$	14	$C::compute_x$	14
$S::main_i$	15	$C::main_i$	15
$S::main_x$	16	$C::main_x$	16

events such that the partial ordering for all internal events is preserved. Next, in the *third step*, method event traces generated from all processes are gathered and merged to a holistic ordered sequence stored in either one or multiple traces. Finally, the *fourth step* takes the query set M and the merged event sequence to compute the impact set of M .

2) *Illustration*: To illustrate the above process flow, consider the example program *E* of Figure 1. DISTEA first instruments *E* and produces instrumented code for both components. Next, suppose the instrumented server and client components S' and C' are deployed on two distributed machines, and S' starts first before a user launches C' . When running concurrently, S' and C' generate two method-event sequences in two separate processes, as listed *in full* in the first and last two columns in Table I, respectively. As is shown, logical clocks are updated upon communication events. For instance, the logical clock of the server process is first updated to 10 upon the event $E_c(C,S)$ originated in the client process, which is greater by 1 than the current logical clock of the client process. Later, the client logical clock is updated to 14 upon $E_c(S,C)$. The internal events are time-stamped by these logical clocks while communication events are not.

Next, suppose the query set $M = \{S::getMax\}$, DISTEA merges event traces of the two processes and, by

inferring impact relations from the timestamped events, it gives $\{S::getMax, S::serve, S::main, C::compute, C::main\}$ as the impact set of M . As is demonstrated, DISTEA can predict impacts across distributed components (processes). For instance, if the developer plans for a change to method `getMax` in the server code, the methods `compute` and `main` in the client code, in addition to the other two server methods, are potentially affected and, thus, need inspections by the developer before applying that change.

C. Analysis Algorithms

1) *Partial Ordering of Internal Events*: Preserving the partial ordering of internal events is at the core of DISTEA, for which two main options exist, both based on a logical notion of time: the LTS approach [32] as described before, and vector clocks [38], [39]. In comparison, LTS is lighter-weight as it just maintains a single counter as the logical clock for each process, while a vector clock keeps an array of clocks for all processes. Therefore, we adopt LTS in this work since it suffices for the current DISTEA design.

Algorithm 1 summarizes in pseudo code the DISTEA algorithm for partially ordering internal events based on the original LTS. The logical clock of the current process C is initialized to 0 upon process start, as is the global variable `remaining`, which tracks the remaining length of data most recently sent by the *sender* process. The rest of this algorithm consists of two parts, which are triggered upon the occurrence of communication events during system executions.

The first part is the runtime monitor `SENDMESSAGE` triggered online upon each message-sending event. The monitor piggybacks (prepends) two extra data items to the original message: the total length sz of the data to send, and the present value of the local logical clock C (of this *sender* process) (lines 2–3); then, it sends out the packed data (line 4).

The second part is the other monitor `RCVMESSAGE`, which is triggered online upon each message-receiving event. After reading the incoming message into a local buffer d (line 6), the monitor decides whether to simply update the size of remaining data and return (lines 7–9), or to extract two more items of data first: the new total data length to read, and the logical clock of the peer *sender* process (lines 10–16). In the latter case, the two items are retrieved, and then removed also, from the entire incoming message (lines 10–11). Next, the remaining data length is reduced by the length of data already read in this event, and the local logical clock (of this *receiver* process) is compared to the received one, updated to the greater, and incremented by 1 (lines 13–15). Lastly, the monitor returns the message as originally sent in the system (i.e., with the prepended data taken away).

To avoid interfering the message-passing semantics of the original system, DISTEA keeps the length of remaining data (with the variable `remaining` in the algorithm) to determine the right timing for logical-clock retrieval. In real-world distributed programs (e.g., Zookeeper [40]), it is common that a *receiver* process may obtain, through several reads, the entire data sent in a single write by its peer *sender* process. For example, a first read just retrieves data length so that an appropriate size of memory can be allocated to take the actual data content in a second read. Therefore, not only is it unnecessary to attempt retrieving the prepended data items (data length and logical clock) in the second read since the first one should have already done so, but also such attempts

Algorithm 1 Monitoring communication events

```
let  $C$  be the logical clock of the current process
remaining = 0 // remained length of data to read
1: function SENDMESSAGE( $msg$ ) // on sending a message  $msg$ 
2:    $sz$  = length of  $sz$  + length of  $C$  + length of  $msg$ 
3:   pack  $sz$ ,  $C$ , and  $msg$ , in order, to  $d$ 
4:   write  $d$ 
5: function RECVMESSAGE( $msg$ ) // on receiving a message  $msg$ 
6:   read data of length  $l$  into  $d$  from  $msg$ 
7:   if remaining > 0 then
8:     remaining -=  $l$ 
9:   return  $d$ 
10:  retrieve and remove data length  $k$  from  $d$ 
11:  retrieve and remove logical clock  $ts$  from  $d$ 
12:  remaining =  $k$  - length of  $k$  - length of  $ts$  -  $l$ 
13:  if  $ts > C$  then
14:     $C = ts$ 
15:  increment  $C$  by 1
16:  return  $d$ 
```

can break the original network I/O protocols.

2) *Monitoring Internal Events*: Impact inference in DISTEA relies on the execution order of methods that is deduced from the timestamps attached to all internal events, for which DISTEA monitors the occurrence of each internal event. However, as proved in [8], recording just the *first* entrance and *last* returned-into (or return) events is equivalent to tracing the full sequence of those events for the dynamic impact analysis in EAS. Similarly, this equivalence also applies in DISTEA. Thus, instead of keeping the timestamp for every internal-event occurrence (as shown in Table I), DISTEA only records two key timestamps for each method m : the one for the first instance of m_e , and the one for the last instance of m_i or m_x , whichever occurred later.

Accordingly, the online algorithm for monitoring internal events uses two counters to record the two key timestamps for each method, similar to what EAS did but different in that it does so *in each process*. Also, we use the per-process logical clock, instead of a global integer as used by EAS, to update the per-method counters during runtime. In the meanwhile, the logical clock C_i of each process P_i is maintained as follows:

- Initialize C_i to 0 upon the start of P_i .
- Increase C_i by 1 upon each internal event occurred in P_i .
- Update C_i upon each communication event occurred in P_i via the two online monitors shown in Algorithm 1.

Finally, for the offline impact computation in DISTEA, the online algorithm here also dumps per-process internal-event sequences (i.e., the two timestamps for each executed method) as traces upon program termination.

3) *Impact Computation*: During system executions, the online internal-event monitoring algorithm generates event traces concurrently and commonly on distributed machines. Since it computes impacts offline, DISTEA gathers these traces to one machine before merging them. Then, from the merged traces, DISTEA computes the impact set of any given query by searching methods that have the execute-after relation with that query according to Equation 3.

IV. IMPLEMENTATION

The DISTEA tool² consists of three main modules: a static analyzer, two sets of runtime monitors, and a post-processor.

A. Static Analyzer

The static analyzer instruments the input program such that all method-execution events are monitored accurately, which is crucial to the soundness and precision of DISTEA. We used Soot [41] for the instrumentation in two main steps. First, DISTEA inserts probes for the three types of internal events in each method, for which we reused relevant modules of DIVER [9]. The second step is to insert probes for communication events, for which DISTEA uses the list L of message-passing APIs, if specified, to identify probe points based on string matching: L includes the prototype of each unique API used in the input system for network I/Os. If L is not specified, a list of basic Java network I/O APIs will be used covering two common means of blocking and non-blocking communications: Java Socket I/O [42] and Java NIO [43].

B. Runtime Monitors

The two sets of runtime monitors implement the two online algorithms: the first for monitoring internal events and the second preserving the partial ordering of them. The first set again reuses relevant parts of DIVER [9]. For the second set, instead of invoking additional network I/O API calls to transfer logical clocks, the monitors take over the original message passing so that they can *piggyback* the two extra data items (i.e., the data length and logical clock) to the original message. To that end, the probes for the monitors *replace* the original network I/O API calls during the instrumentation.

That is, the extra data items are carried on by the original message passing. Our experience suggested that this piggyback strategy is more viable than inserting additional calls, especially when dealing with selector-based non-blocking communications [33]. For instance, the ShiVector tool in [44], which adopted the latter, was unable to work with two of our subject programs (NioEcho and ZooKeeper). One reason as we found is that, for a pair of an original call and the corresponding additional call, the two messages may not be read in the same order by the *receiver* process as in which they are sent by the *sender* process. As a result, an original message-receiving call may encounter extraneous data in the message hence the violation of original network I/O semantics.

C. Post-processor

The post-processor is the module that actually answers impact-set queries. To that end, it collects distributed traces through a helper script which passes per-process traces to the offline impact-computation algorithm. To compute impact sets, the post-processor retrieves the partial ordering of internal events by just comparing their timestamps.

V. EMPIRICAL EVALUATION

To evaluate our approach, we conducted an empirical study to answer the following three research questions:

- **RQ1** How effective is DISTEA in predicting impacts relative to existing alternative options?
- **RQ2** How does impacts within processes compare with impacts across process boundaries?
- **RQ3** How efficient and scalable is DISTEA in terms of the time and storage overheads it incurs?

The main goal of this evaluation was to investigate the effectiveness (RQ1) and efficiency (RQ3) of DISTEA. We also intended to examine the composition of DISTEA impact sets concerning how impacts propagate within and across distributed components (processes) (RQ2).

²Download of the entire package is available at <http://nd.edu/~hcai/distea>.

TABLE II: STATISTICS OF EXPERIMENTAL SUBJECTS

Subject	#SLOC	#Methods	Inputs (size and type)	#Queries
MultiChat (r5)	470	37	1 integration test	25
NioEcho (r69)	412	27	1 integration test	26
ZooKeeper (v3.4.6)	62,450	4,813	1 integration test	749
			1 system test	817
			1 load test	798
			195 unit tests	2,780
Voldemort (v1.9.6)	163,601	17,843	1 integration test	2,048
			1 system test	1,056
			1 load test	1,323
			9 unit tests	3,421

A. Experiment Setup

We evaluated DISTEA on four distributed Java programs, as summarized in Table II. The size of each subject is measured by the number of non-comment non-blank source lines of code (#SLOC), and number of methods defined in the subject, both in Java, that we actually analyzed. The last two columns list the input sets we used in our study, including the type and size of each set, and the number of methods (#Queries) covered in that set that we all used as impact-set queries.

MultiChat [45] is a chat application where multiple clients exchange messages via a server broadcasting the message sent by one client to all others. NioEcho [46] is an echo service via which the client just gets back the same message as it sends to the server. ZooKeeper [40], [47] is a coordination service for distributed systems to achieve consistency and synchronization. Voldemort [48] is a distributed key-value storage system used at LinkedIn. The first two use only Socket I/O and Java NIO, respectively, and the last two used both. For all subjects, we checked out from their official online repositories for the latest versions or revisions as shown in (the parentheses of) Table II.

We chose these subjects such that a variety of program sizes, application domains, and uses of both blocking and non-blocking I/Os are all considered; we chose the input sets to cover different types of inputs when possible, including system test, integration test, load test, and unit test. Except for the integration tests, other types of inputs come with the subjects as integral parts. For unit tests, we used only those leading to multiprocess executions from the full original sets.

For each subject, we created the single integration test in which we started one server process and one client process on two separated machines and manually performed client operations that cover basic server functionalities. Specifically, for MultiChat and NioEcho, the client requests were sending random text messages; for ZooKeeper, the client operations were, in order: create a node, look up for it, check its attributes, change its data association, and delete it; for Voldemort, the operations were, also in order: add a key-value pair, query the key for its value, delete the key, and retrieve the pair again.

B. Experimental Methodology

To the best of our knowledge, there are no other dynamic impact analysis techniques for distributed systems in the literature. We could not compare to slicing techniques either as we are not aware of such slicers readily available to us and working with real-world distributed systems like our subjects, while developing one would require considerable efforts.

Therefore, we assume two possible alternatives to DISTEA: ignoring impacts outside the process where the query first executed (referred to as *local process*, versus all others as *remote processes*); taking all methods executed in remote processes as impacted. In contrast, the latter (i.e., method-level coverage, referred to as *MCov*) is safe

hence potentially a more practical option in most cases. In fact, *MCov* is an easy adaptation for distributed systems from COVERAGEIMPACT [34], a major existing option for centralized programs. Thus, we consider *MCov* as the baseline technique and the covered (executed) sets of methods as baseline impact sets. We refer to impacts in local and remote processes as *local impacts* and *remote impacts*, respectively.

For every method of each subject as the query, we measure the effectiveness of DISTEA by comparing the impacts it predicted to those given by *MCov* in terms of impact-set size ratios, and examine the composition of the impact set concerning its two subsets: *local impact set* and *remote impact set*, while also analyzing their intersection, referred to as *common impact set*. Accordingly, we measure too the effectiveness of DISTEA with respect to these subsets relative to the corresponding *MCov* results.

When computing the impact set for a query, for each type of inputs, we take the union of per-input impact sets of all inputs of that type since each type is usually intended to represent a different operational profile of the system under analysis. Finally, beside the impact querying time, we report the static-analysis and runtime costs of DISTEA, and storage costs of event traces, together as efficiency metrics. All machines used in our experiments are Linux workstations with an Intel i5-2400 3.10GHz processor and 8GB DDR2 RAM.

C. Results and Analysis

1) *RQ1: Effectiveness*: Figure 3 shows the main effectiveness results, with one plot depicting the data distribution for each subject and input type, shown as the plot title (hereafter, the input type is omitted for the two small subjects as only integration test is used for them). Each plot includes three box plots showing that data distribution for one of three categories (on x axis): the holistic impact set (*all*) and its two subsets (*local* and *remote*), with each data point indicating the effectiveness metric (on y axis) for one query.

The results show that DISTEA is constantly more effective than *MCov*, reducing the impact sets of the latter by 15% to over 95%. Compared to the small subjects, the two large subjects see noticeably better effectiveness, possibly because the few methods in the small ones tend to all focus on a single task hence have more methods executed after any other methods. Also, the ratios with respect to *all* impact sets are always higher than those to the two subsets. The reason is that the two subsets from both approaches have substantial intersections, while the ones from DISTEA are consistently smaller than those from *MCov*.

Overall, as shown by the complementary results, the mean effectiveness, in Table III (left four columns), DISTEA reports on average only 64% of the impacts produced by *MCov*. In particular, the reductions in remote impact sets are even higher, by 56% on average and well above 50% in most individual cases. This implies that, relative to the baseline, developers can save the time that would be spent on inspecting more than half of the impacts propagated in remote processes.

2) *RQ2: Impact-set Composition*: Figure 4 plots the impact-set composition for each individual query numbered on the y axis, where the x axis indicates the percentage of three complementary sets, *local*, *remote*, and *common*, for each subject and input-set type. The common sets have been removed from the local and remote subsets, but in this figure only, to help clarify the composition.

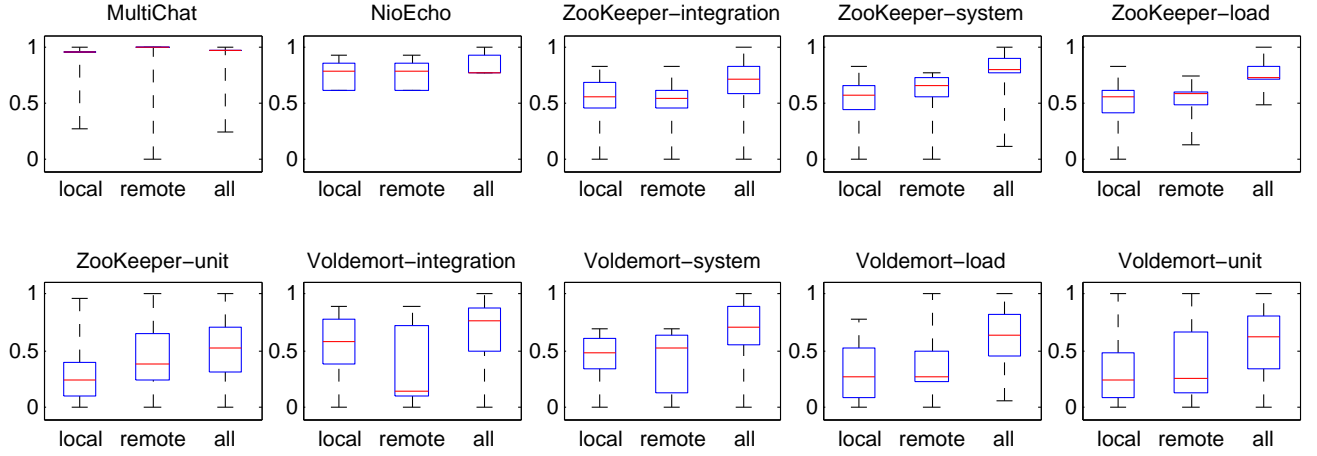


Fig. 3: Effectiveness of DISTEA expressed as the ratios (y axes) of its per-query impact-set sizes, including those of the local and remote subsets (x axes), versus *MCov* as the baseline, for each subject and input-set type (atop each plot as the title).

A first observation is that remote impact sets dominate corresponding holistic impact sets in way most cases. For one thing, this might explain the mostly higher impact-set size ratios for remote impacts than for local ones, as seen in Figure 3. For another, the contrast in size between the two subsets suggests that impacts can propagate much more largely to remote processes than in local ones in distributed programs.

Another finding is that, for almost all queries, there were methods executed after the query in both local and remote processes. This implies that in distributed systems, components often share common functionalities. Moreover, the sizes of common impact sets could be a metric of functional overlapping and code reuse among components of distributed systems. Also, the figure shows that the strength of this metric seems to continuously increase with the system size.

3) *RQ3: Efficiency*: Table III lists all relevant costs of DISTEA for this study, including the runtime of static analysis, runtime overhead measured as ratios of the runtime of the original program (*Normal run*) over the instrumented one (*Instrumented run*), and impact querying time.

The static analysis took longer for larger subjects as expected, yet still within 2.2 minutes even on the largest system Voldemort. Note that this is a one-time cost (for the single program version analyzed by DISTEA at least) as the instrumented code can be executed on any inputs and used for computing any queries afterwards. Runtime and querying costs are constantly correlated to subject sizes as well as the sizes of input sets, with the worst case seen by ZooKeeper on its unit-test input set, which is by far the largest among all subjects and inputs studied. Nevertheless, the runtime overhead is at worst 26% and the longest querying time is in 15 seconds.

Storage costs are also tightly connected to the number of inputs in addition to subject sizes, of which the largest is 188MB for the 195 traces of ZooKeeper. In other cases, this cost is at most 7MB, with an overall average less than 20MB.

In all, the results suggest that DISTEA is highly efficient in both time and space dimensions, and that it seems to be readily scalable up to large systems. As shown in the bottom row of the table, it costs on average less than one minute for static analysis and a couple seconds for computing the impact set per query, with the mean runtime overhead of about 10%.

D. Threats to Validity

The main threat to *internal* validity lies in possible implementation errors in DISTEA and experiment scripts.

To reduce this threat, we did a careful code review for our tools and used the two small subjects to manually validate their functionalities and analysis results. An additional such threat concerns about possible missing (remote) impacts due to network I/Os that were not monitored at runtime. However, we checked the code of all subjects and confirmed that they only used the most common message-passing APIs monitored by our tool with respect to their input sets that we studied.

The main threat to *external* validity is that our study results may not generalize to other distributed programs and input sets. In this study, we considered only limited number of subjects, which may not represent all real-world programs, and only subsets of inputs, which do not necessarily represent all behaviours of the studied programs. To reduce this threat, we have chosen programs of various sizes and application domains, including the two enterprise systems in different areas. In addition, we considered different types of inputs, including integration, system, load, and unit tests. Most of these tests came as part of the subjects except for the integration tests, which we created according to the official online design documentation of these programs.

The main threat to *construct* validity is the metrics used for the evaluation. Without directly comparable peer techniques in the literature, we assumed that developers would use coverage-based approach like *MCov*, as a representative alternative to DISTEA, to narrow down the search space of potential impacts in the context of distributed executions. To mitigate this threat, we examined the composition of each impact set and analyzed the effectiveness with respect to its local and remote subsets in addition to that of the holistic impact set to help demonstrate the usefulness of DISTEA.

Finally, a *conclusion* threat concerns about the data points analyzed: We applied the statistical analyses only to methods for which impact sets could be queried (i.e., methods executed at least once). Also, the present study only considered potential changes in single methods for each query, while in practice developers may plan for changes in multiple methods at a time, which may lead to different results. To minimize this threat, we adopted this strategy for all experiments and calculated the experimental metrics for every possible query.

VI. CASE STUDIES

To further investigate the effectiveness and, more important, the practical utility of DISTEA, we conducted two case studies. In contrast to the foregoing empirical evaluation,

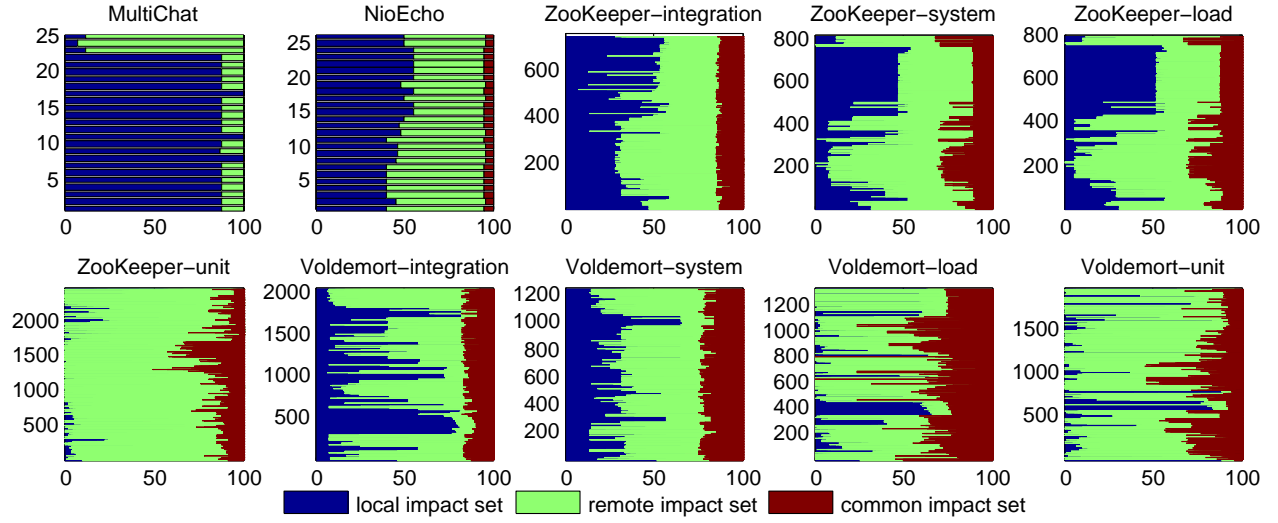


Fig. 4: Composition of impact sets given by DISTEA for all queries (y axes) expressed as the percentages (x axes) of local, remote, and common impact sets in the whole impact set per query, for each subject and input-set type (atop each plot).

TABLE III: MEAN EFFECTIVENESS, TIME-COST BREAKDOWN, AND STORAGE COSTS OF DISTEA

Subject & input	Mean impact-set size ratios			Time costs (ms)					Storage costs (KB)
	Local	Remote	All	Static analysis	Normal run	Instrumented run	Runtime overhead	Querying (with stdev)	
MultiChat	85.33%	85.03%	86.08%	12,817	5,461	5,738	5.07%	3 (2)	6
NioEcho	77.07%	76.42%	83.73%	13,365	3,213	3,623	12.76%	3 (3)	4
ZooKeeper-integration	55.85%	53.83%	70.73%	39,124	37,239	38,416	3.16%	10 (3)	96
ZooKeeper-system	53.54%	62.22%	81.17%		15,385	18,578	20.75%	16 (8)	136
ZooKeeper-load	50.14%	55.07%	76.17%		94,187	98,930	5.04%	15 (7)	140
ZooKeeper-unit	28.56%	43.26%	51.44%		1,109,146	1,370,143	23.53%	14,619 (87,056)	188,804
Voldemort-integration	55.86%	40.12%	69.72%	132,536	17,755	18,697	5.31%	27 (9)	312
Voldemort-system	45.71%	42.36%	67.12%		11,136	12,253	10.03%	19 (7)	196
Voldemort-load	30.90%	37.50%	61.68%		21,066	21,253	0.89%	122 (190)	776
Voldemort-unit	31.77%	37.55%	57.50%		132,676	167,861	26.52%	403 (835)	6,984
Overall average	41.42%	43.99%	63.88%	49,460.5	144,726.4	175,549.2	11.31%	3,228.9 (40,752.1)	19,745.4

the case studies were focused on sample subjects and inputs against a small number of queries for in-depth examination.

A. Study I: Precision and Usefulness of DISTEA Impact Sets

1) *Methodology*: As we discussed earlier, DISTEA can be imprecise. Yet, currently there is no automatic means available for us to assess exactly how imprecise it would be. Thus, in our first case study, we investigate the precision of DISTEA. To that end, we randomly chose two queries from a small subject MultiChat and three from a large one Voldemort, and picked an input set for each also randomly. We then manually determine the ground-truth impact set of the chosen queries according to our understanding of the system’s runtime behaviour with respect to the input. Since the manual inspection is exhaustive, we limited our choices of queries and inputs to those for which the DISTEA impact sets had no more than 50 methods.

2) *Results*: Table IV lists the results for the five cases we studied. For each case, the table summarizes the impact-set sizes (IS) from DISTEA, manual inspection, and the baseline approach ($MCov$), all separately for the two subsets (*local* and *remote*). The numbers in parentheses are the precision of DISTEA (recall was constantly 100%).³

In most of these cases, a considerable portion of the impact sets was methods executed after but not to be impacted by the query, as we expected. For instance, the first query in MultiChat was the `run` method of the main client thread, which executed at the end of the client process to iteratively send user inputs to the server. As a result, in the local process the query could only impact itself; and of 13 methods executed

TABLE IV: RESULTS FOR FIVE CASES OF USING DISTEA

Subject & input	DISTEA IS (precision)		Manual true IS		MCov IS	
	local	remote	local	remote	local	remote
MultiChat	1 (100%)	13 (69.2%)	1	9	3	21
MultiChat	13 (76.9%)	2 (50%)	10	1	22	3
Voldemort-system	4 (100%)	23 (56.5%)	4	13	740	809
Voldemort-system	3 (33.3%)	0 (-)	1	0	811	440
Voldemort-load	13 (46.1%)	41 (41.4%)	6	17	288	500
Overall average	6.8 (71.2%)	15.8 (51.7%)	4.7	8	373	354.6

after it in the server process, four were false positives as they just dealt with network connections independently of any specific message received from clients.

For another example, the last query in Voldemort, an error-handling utility method, is defined in a common module executed by both the Voldemort master and its clients. Among six common impacts reported by DISTEA, only two were possibly to be impacted by the query. The other four, along with three of the seven unique impacts in the local process, never involved error handling. These methods, as 24 out of 41 methods devoted to network service maintenance only in the remote process, were falsely reported as impacted.

In all, DISTEA had an overall average precision of 56.9% for the five randomly selected queries, and for remote impact sets only the number was 51.7%. These are very close to the precision of EAS obtained from an extensive study using various types of changes in [36]. In contrast, precision of the local impact sets was considerably higher, not only on overall average but constantly in every single case. This may be due to the even looser coupling, via message passing only, among methods across components than within components.

³Details on the results are at <http://nd.edu/~hcai/distea/casestudy1.html>.

Since we only studied five cases, these results are by no means conclusive. Yet, it seems to suggest that developers could expect impact sets close to 60% precise from DISTEA in an average case. Note that although this precision may not be sufficient in some situations, such as when reported impact sets are extremely large, DISTEA is reasonably effective and useful with respect to the much larger sets of covered methods (*MCov* results). For instance, checking on average 16 methods only with DISTEA, instead of 354 with *MCov*, for impacts propagated to remote components implies significant reduction in impact-inspection efforts for the studied cases.

B. Study II: Utility for Distributed-Program Understanding

1) *Methodology*: Dynamic analysis is an important means for program comprehension based on concrete executions, a process on which developers often spend a great deal of effort [49]. However, understanding distributed system executions is a challenging task because of the complex interactions among concurrent component executions in such systems [50], and even more so in the presence of selector-based non-blocking communications [33].

Thus, our second case study aimed to explore the utility of DISTEA for program comprehension of distributed systems. In particular, we intended to see if DISTEA can help understand the interface between distributed components and interprocess communications (IPC) among them. For this study, we chose NioEcho and ZooKeeper, both of which utilize selector-based network I/Os (Java NIO [33]) for IPC. For each of these two subjects, without prior knowledge about internals of either, we first picked a few important-looking queries (simply based on names) from each component, and then executed the instrumented program on the same integration test as used in the empirical study. Next, we took the DISTEA impact sets of the selected queries to learn about its runtime IPC semantics.

2) *Results*: For NioEcho, despite of its small source size and simple high-level functionality, the non-blocking IPCs between the server and client made it much harder than expected to fully understand the program, interactions between the two components in particular, by just reading its source code. To use our tool, we picked two queries from the client and three from the server that all *looked* closely relevant to messaging. It turned out that using DISTEA was quite instrumental in this case: the local and remote impacts, when listed together in the ascending order of associated timestamps, clearly show how the client initiates a response handler for a message before sending it out and, before it gets to wait for response in the handler by checking the selector it registered, the server already received the message and started its echo service, after which the client reads the server response.

In the case of ZooKeeper, given the large size and complexity of the entire system, we targeted only the particular IPCs with respect to one fundamental operation `getData` [47]. We started with the entry methods of both the server and client modules, and then by searching in their impact sets we located one most relevant-looking query from the client. Next, examining local and remote impacts of that query let us identify the major transaction steps: The client first prepares a data request using an external library and then forwards the request to a client thread which spawned another child thread to actually connect to the server; next, when the client proceeds with some bookkeeping routines while waiting for response, the execute-after sequence (DISTEA results) in

the remote process shows that the server has accepted the request and initiated a thread to access a database, and then started another thread to send the retrieved data back, followed by the client's taking the response and processing the data.

In both cases, we also found the ordering of impacts (by their timestamps) fairly useful for following component-level interactions step by step, and that the common impacts, which reveal code reuse among components, also facilitate the comprehension process. On the other hand, however, we noticed that navigating in the textual impact sets can be tedious when the results become large, for which some effective visualizations [50], [51] would be helpful. One possible solution, for example, is to visualize the partially ordered impacts as an *interprocess* call graph, which can complement or collaborate with other distributed-program comprehension approaches, such as space-time diagrams and communicating finite state machines [18].

In sum, while the results of this exploratory study may not generalize to other cases and systems, our experience suggests that DISTEA can help users understand distributed programs and their executions. Note that the main source of this benefit lies in the remote impacts DISTEA produces, which tell about the interactivity among distributed components.

VII. DISCUSSION

The core technique of DISTEA is to partially order distributed method-execution events to discover execute-after relations among methods in multiple concurrent processes. And we have demonstrated that the technique can be an important step for effectively evolving distributed systems. On the other hand, its conservative nature, while makes it safe modulo the concrete executions utilized, can also lead to false positives. Nonetheless, DISTEA will be a practically useful option for developers since it provides rapid, although possibly rough, results [52]. DISTEA is highly efficient, whereas a more accurate analysis would need to trade efficiency for precision if remaining sound. Also, developers actually need multiple levels of cost-effectiveness tradeoffs for impact analysis [10].

A few other limitations exist. First, the present tool might not immediately fully work for arbitrary distributed systems, because it now considers only the two common cases of message-passing APIs by default and, even with the user-specified list of all such APIs, the text matching by API prototypes DISTEA uses for locating instrumentation points may cause incomplete communication-event monitoring. However, this is the limitation of our current implementation, not of the technique. In addition, the instrumentation for monitoring method events and extra network traffics for exchanging logical clocks may affect the performance of original systems [18], although we expect such effects to be minor in general according to our studies.

VIII. RELATED WORK

Three main categories of previous work are related to ours: dynamic impact analysis, dependence analysis of concurrent programs, and logging and timing for distributed systems.

A. Dynamic Impact Analysis

The execute-after-sequences (EAS) approach [8] which partially inspired DISTEA is a performance optimization of its predecessor PATHIMPACT [7]. Many other dynamic impact analysis techniques also exist [6], aiming at improving precision [9], [53]–[55], recall [56], efficiency [57], [58], and

cost-effectiveness [10] over PATHIMPACT and EAS. However, these techniques did not address distributed or multiprocess programs that we focus on in this work.

Two recent advances in dynamic impact analysis, DIVER [9] and the multivariate framework in [10], utilize hybrid program analysis to achieve higher precision and more flexible cost-effectiveness options over EAS-based approaches, but still focus on centralized programs. As a first step, DISTEA sacrifices imprecision for high efficiency. However, it would be interesting to adopt hybrid approaches for distributed systems too. For instance, among other improvements, one may be to immediately gain better analysis precision by first using static dependencies to prune false-positive impacts *within* each process, as DIVER did, and then propagating impacts across process boundaries by means of the Lamport timestamps as used in this work. More aggressive pruning may also be promising if leveraging more and/or finer interprocess dependencies, such as communication dependencies and synchronization dependencies [20], [22], as exploited by distributed-program slicing techniques [23].

B. Dependence Analysis of Concurrent Programs

Using fine-grained dependency analysis, a large body of work attempted to extend traditional slicing algorithms to concurrent programs [13]–[15], [59], [60] yet mostly focusing on centralized, and primarily multithreaded, ones. For those programs, traditional dependence analysis was extended to handle additional dependencies due to shared variable accesses, synchronization, and communication between threads and/or processes (e.g., [13], [59]). While DISTEA also handles multithreaded programs, it targets multiprocess ones running on distributed machines, and aims at lightweight impact analysis instead of fine-grained slicing.

For systems running in multiple processes where interprocess communications are realized via socket-based message passing, an approximation for static slicing was discussed in [13]. Various dynamic slicing algorithms have been proposed too, earlier for procedural programs only [12], [19]–[21], [61] and recently for object-oriented software also [22]–[24]. And a more complete and detailed summary of slicing techniques for distributed programs can be found in [60] and [23]. Although these slicing algorithms were rarely evaluated against large real-world distributed systems, it can be anticipated that they would face scalability issues with large systems based on the limited empirical results they reported and the heavyweight nature of their design.

In contrast to these fine-grained (statement-level) analysis, DISTEA aims at a highly efficient method-level dynamic impact analysis that can readily scale up to large distributed programs. A few more static analysis algorithms for distributed systems exist as well but focus on other (special) types of systems, such as RMI-based Java programs [62] and Android applications [25], different from the common type of distributed systems [35] DISTEA addresses.

At coarser levels, other researchers resolve dependencies in distributed systems too but for different purposes such as enhancing parallelization [63], system configuration [64], and high-level system modeling [18], [44]. A static analysis, LSME [31] extracts inter-component dependencies due to implicit invocations, but it is both imprecise and unsound [17], [27]. In [17], another static analysis is proposed to infer inter-component dependencies based on messaging-interface

matching. In contrast, DISTEA performs code-based analysis while providing more focused impacts relative to concrete program executions than static-analysis approaches.

An impact analysis dedicated to distributed systems, Helios [27] can predict impacts of potential changes to support evolution tasks for DEBS. However, it relies on particular message-type filtering and manual annotations in addition to a few other constraints. Although these limitations are largely lifted by its successor Eos [17], both approaches are *static* and limited to DEBS only, as is the latest technique [26] which identifies impacts based on change-type classification yet ignores intra-component dependencies hence provides merely incomplete results. While sharing similar goals, DISTEA targets a broader range of distributed systems than DEBS using *dynamic* analysis and without relying on special source-code information (e.g., interface patterns) as those techniques do.

C. Logging and Timing for Distributed Systems

To facilitate high-level understanding of distributed systems and executions, techniques like logging and mining runtime logs [18], [30] infer inter-component interactions using textual analysis of system logs, relying on the availability of particular data such as informative logs and certain patterns in them. DISTEA also utilizes similar information (i.e., the Lamport timestamps) but infers the happens-before relations among method-execution events mainly for code-level impact analysis. Also, DISTEA automatically generates such information it requires rather than relying on existing information in the original programs (e.g., logging statements).

The Lamport timestamp used by DISTEA is closely related to the vector clocks [38], [39] used by other tools, such as ShiVector [44] for ordering distributed logs and Poet [51] for visualizing distributed systems executions. While we could utilize vector clocks also, we chose the Lamport timestamp as it is lighter-weight and simpler yet well suffices for this work. In addition, unlike ShiVector, which requires accesses to source code and recompilation using the AspectJ compiler, DISTEA does not have such constraints as it works on bytecode.

IX. CONCLUSION

Components in distributed systems usually run concurrently in separated processes and communicate via socket-based message passing without explicitly invoking or referencing each other. In consequence, existing dynamic impact analysis, which relies on explicit invocations (dependencies), tends to be either entirely inapplicable or at best quite ineffective to use for those systems.

We presented DISTEA, a dynamic impact analysis for evolving common-type distributed systems. By leveraging lightweight monitoring of method-execution events and partially ordering those events during concurrent multiprocess executions, DISTEA can safely predict potential impacts of any query both within and across all processes, supporting efficient dynamic impact analysis of distributed systems. DISTEA has been implemented for Java and is publicly available for download. Through an empirical evaluation and two case studies on four distributed Java programs, including two large real-world enterprise systems, we have shown the high efficiency of DISTEA and its superior effectiveness over existing alternatives, and also illustrated its benefits for understanding distributed systems and executions. Overall, DISTEA offers a promising option to developers for maintaining and evolving distributed systems.

REFERENCES

- [1] V. Rajlich, "Changing the paradigm of software engineering," *Communications of the ACM*, vol. 49, no. 8, pp. 67–70, 2006.
- [2] S. A. Bohnner and R. S. Arnold, *An introduction to software change impact analysis*. Software Change Impact Analysis, IEEE Comp. Soc. Press, pp. 1–26, Jun. 1996.
- [3] P. Rovegard, L. Angelis, and C. Wohlin, "An empirical study on views of importance of change impact analysis issues," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 516–530, 2008.
- [4] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: an exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 51:1–51:11.
- [5] V. Rajlich, "Software evolution and maintenance," in *Proceedings of the Conference on the Future of Software Engineering*, 2014, pp. 133–144.
- [6] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, pp. 613–646, 2013.
- [7] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2003, pp. 308–318.
- [8] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proc. of Int'l Conf. on Softw. Eng.*, May 2005, pp. 432–441.
- [9] H. Cai and R. Santelices, "Diver: Precise dynamic impact analysis using dependence-based trace pruning," in *Proceedings of International Conference on Automated Software Engineering*, 2014, pp. 343–348.
- [10] —, "A Framework for Cost-effective Dependence-based Dynamic Impact Analysis," in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015, pp. 231–240.
- [11] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 245–254.
- [12] D. Goswami and R. Mall, "Dynamic slicing of concurrent programs," in *IEEE International Conference on High Performance Computing*, 2000, pp. 15–26.
- [13] J. Krinke, "Context-sensitive slicing of concurrent programs," in *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5. ACM, 2003, pp. 178–187.
- [14] J. Xiao, D. Zhang, H. Chen, and H. Dong, "Improved program dependence graph and algorithm for static slicing concurrent programs," in *Advanced Parallel Processing Technologies*. Springer, 2005, pp. 121–130.
- [15] D. Giffhorn and C. Hammer, "Precise slicing of concurrent programs," *Automated Software Engineering*, vol. 16, no. 2, pp. 197–234, 2009.
- [16] K. Jayaram and P. Eugster, "Program analysis for event-based distributed systems," in *Proceedings of the 5th ACM International Conference on Distributed Event-Based System*. ACM, 2011, pp. 113–124.
- [17] J. Garcia, D. Popescu, G. Safi, W. G. Halfond, and N. Medvidovic, "Identifying message flow in distributed event-based systems," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 367–377.
- [18] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with csight," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 468–479.
- [19] B. Korel and R. Ferguson, "Dynamic slicing of distributed programs," *Applied Mathematics and Computer Science*, vol. 2, no. 2, pp. 199–215, 1992.
- [20] M. Kamkar and P. Krajina, "Dynamic slicing of distributed programs," in *Software Maintenance, 1995. Proceedings., International Conference on*. IEEE, 1995, pp. 222–229.
- [21] E. Duesterwald, R. Gupta, and M. Soffa, "Distributed slicing and partial re-execution for distributed programs," in *Languages and Compilers for Parallel Computing*. Springer, 1993, pp. 497–511.
- [22] D. P. Mohapatra, R. Kumar, R. Mall, D. Kumar, and M. Bhasin, "Distributed dynamic slicing of java programs," *Journal of Systems and Software*, vol. 79, no. 12, pp. 1661–1678, 2006.
- [23] S. S. Barpanda and D. P. Mohapatra, "Dynamic slicing of distributed object-oriented programs," *IET software*, vol. 5, no. 5, pp. 425–433, 2011.
- [24] S. Pani, S. M. Satapathy, and G. Mund, "Slicing of programs dynamically under distributed environment," in *Proceedings of International Conference on Advances in Computing*. Springer, 2012, pp. 601–609.
- [25] D. Outeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *USENIX Security 2013*, 2013.
- [26] S. Tragatschnig, H. Tran, and U. Zdun, "Impact analysis for event-based systems using change patterns," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 763–768.
- [27] D. Popescu, J. Garcia, K. Bierhoff, and N. Medvidovic, "Impact analysis for distributed event-based systems," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. ACM, 2012, pp. 241–251.
- [28] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Springer, 2006, vol. 1.
- [29] P. Eugster and K. Jayaram, "EventJava: An extension of java for event correlation," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 570–594.
- [30] J.-G. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured logs analysis," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 91–96, 2010.
- [31] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 3, pp. 262–292, 1996.
- [32] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
- [33] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weigl, and M. Yamamoto, "Software model checking for distributed systems with selector-based, non-blocking communication," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, 2013, pp. 169–179.
- [34] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proc. of 9th European Softw. Eng. Conf. and 10th ACM SIGSOFT Symp. on the Foundations of Softw. Eng.*, Helsinki, Finland, september 2003, pp. 128–137.
- [35] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, 2011.
- [36] H. Cai and R. Santelices, "A Comprehensive Study of the Predictive Accuracy of Dynamic Change-Impact Analysis," *Journal of Systems and Software (JSS)*, vol. 103, pp. 248–265, 2015.
- [37] A. Orso, T. Apiwattanapong, J. B. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proc. of 26th IEEE and ACM SIGSOFT Int'l Conf. on Softw. Eng. (ICSE 2004)*, Edinburgh, Scotland, may 2004, pp. 491–500.
- [38] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, 1988, pp. 56–66.
- [39] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.
- [40] Apache, "ZooKeeper," <https://zookeeper.apache.org/>, 2015.
- [41] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "Soot - a Java Bytecode Optimization Framework," in *Cetus Users and Compiler Infrastructure Workshop*, Oct. 2011.
- [42] Oracle, "Java Socket I/O," <http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html>, 2015.
- [43] —, "Java NIO," <http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html>, 2015.
- [44] J. Abrahamson, I. Beschastnikh, Y. Brun, and M. D. Ernst, "Shedding light on distributed system executions," in *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 598–599.
- [45] GoogleCode, "MultiChat," <https://code.google.com/p/multithread-chat-server/>, 2015.

- [46] SourceForge, “NioEcho,” <http://rox.xmlrpc.sourceforge.net/niotut/index.html#Thecode>, 2015.
- [47] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems,” in *USENIX Annual Technical Conference*, vol. 8, 2010, p. 9.
- [48] Apache, “Voldemort,” <https://github.com/voldemort/voldemort>, 2015.
- [49] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, “A systematic survey of program comprehension through dynamic analysis,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, 2009.
- [50] J. Moc and D. A. Carr, “Understanding distributed systems via execution trace data,” in *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, 2001, pp. 60–67.
- [51] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten, “Poet: Target-system independent visualizations of complex distributed-application executions,” *The Computer Journal*, vol. 40, no. 8, pp. 499–512, 1997.
- [52] D. Jackson and M. Rinard, “Software analysis: A roadmap,” in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 133–145.
- [53] B. Breech, M. Tegtmeier, and L. Pollock, “Integrating influence mechanisms into impact analysis for increased precision,” in *Int’l Conf. on Softw. Maint.*, 2006, pp. 55–65.
- [54] L. Huang and Y.-T. Song, “Precise dynamic impact analysis with dependency analysis for object-oriented programs,” in *Int’l Conf. on Software Engineering Research, Management & Applications*, 2007, pp. 374–384.
- [55] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio, “On the precision and accuracy of impact analysis techniques,” in *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science*, 2008, pp. 513–518.
- [56] M. C. O. Maia, R. A. Bittencourt, J. C. A. de Figueiredo, and D. D. S. Guerrero, “The hybrid technique for object-oriented software change impact analysis,” in *Euro. Conf. on Software Maintenance and Reengineering*, 2010, pp. 252–255.
- [57] B. Breech, A. Danalis, S. Shindo, and L. Pollock, “Online impact analysis via dynamic compilation technology,” in *Proceedings of IEEE International Conference on Software Maintenance*, 2004, pp. 453–457.
- [58] B. Breech, M. Tegtmeier, and L. Pollock, “A comparison of online and dynamic impact analysis algorithms,” in *Euro. Conf. on Softw. Maint. and Reengg.*, 2005, pp. 143–152.
- [59] M. G. Nanda and S. Ramesh, “Interprocedural slicing of multithreaded programs with applications to java,” *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 6, pp. 1088–1144, Nov. 2006.
- [60] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, “A brief survey of program slicing,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.
- [61] J. Cheng, “Dependence analysis of parallel and distributed programs and its applications,” in *Advances in Parallel and Distributed Computing, 1997. Proceedings.* IEEE, 1997, pp. 370–377.
- [62] M. Sharp and A. Rountev, “Static analysis of object references in rmi-based java software,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 9, pp. 664–681, 2006.
- [63] K. Psarris and K. Kyriakopoulos, “An experimental evaluation of data dependence analysis techniques,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 3, pp. 196–213, 2004.
- [64] F. Kon and R. H. Campbell, “Dependence management in component-based distributed systems,” *IEEE concurrency*, vol. 8, no. 1, pp. 26–36, 2000.